ENGR-UH 3511

Computer Organization and Architecture

Name: Nishant Aswani

Net ID: nsa325

Assignment Title: Lab 6

Microprocessor Design and Verilog HDL: Part 4

Nishant Aswani, nsa325@nyu.edu

Computer Organization and Architecture(ENGR-UH 3511), Instructor Cristoforos Vasilatos

1 Introduction

Pipelining a CPU improves the instructions throughput in a CPU. By breaking down the CPU's actions into five stages, using registers, multiple instructions may be processed simultaneously, at different stages.

The following lab uses Verilog to continue an implementation of a 32-bit MIPS CPU. In this lab, the MIPS CPU is broken down into five stages: Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write Back.

2 Methodology

Following the diagrams, multiple pipeline registers were added as intermediates between the existing components. For ease of use, a separate register was used for each of the inputs. Clocking was then removed from all components but the pipeline registers and the program counter (PC).

Moreover, an implementation, not provided in the diagrams, for jump functionality was developed provided the constraints.

As in the previous lab, the `sample.s` file was converted into binary for the instruction memory.

```
mem[0] <= 32'b00000000000000000000000000000000; // EMPTY
mem[1] <= 32'b00100001000010000000000000000010; // addi $t0 $t0 2
mem[2] <= 32'b00100001010010100000000000000010; // addi $t2 $t2 2
mem[3] <= 32'b00000001000010100100000000100000; // add $t0 $t0 $t2
mem[4] <= 32'b00100001001010010000000000000001; // addi $t1 $t1 1
mem[5] <= 32'b10101101001010000000000000000000; // sw $t0 0($t1)
mem[6] <= 32'b10001101001010100000000000000000; // lw $t2 0($t1)
mem[7] <= 32'b00000001000010010100000000100010; // sub $t0 $t0 $t1
mem[8] <= 32'b00010001000010010000000000000001; // beq $t0 $t1 end
mem[9] <= 32'b00001000000000000000000000000111; // j loop
```

Refer to lab 5 for how the assembly instruction labels were resolved in conversion to binary.

3 Results

3.1 Program Counter

The program counter operates at the positive edge of the clock. The program counter shows the eventual failure the CPU runs into: there is no looping back to a previous instruction as was the case in the previous lab, the reason for which will be explained later below.

Figure 1: Wave output of program counter

3.2 Staggered Output Demonstration

The waveform below demonstrates the output delay of the `PCnext` signal as a way to show that the pipeline registers function as required.

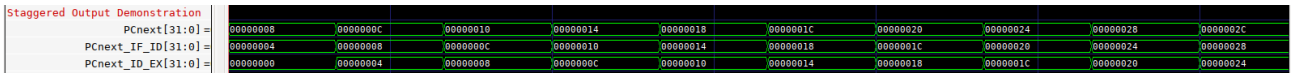

Figure 2: Wave output PCnext at various stages

3.3 Pipelining Stage 1

The outputs of the PC adder and Instruction Memory (IM) are fed as inputs into the first pipeline register.

Figure 3: Wave output of stage 1

3.4 Pipelining Stage 2

Comparing PCnext and IMout in the screenshot below to the signals above, we see that there is a stage delay. In the screenshot above, `PCnext` is `0x8` on the first cycle, while `PCnext-IF-ID` is `0x4` in the screenshot below.

Aside from the delays, we can also clearly the five stage pipeline begin to play out. In the first cycle, the `readRegisterOne` and `readRegisterTwo` signals are blank, implying the `IF` stage. In the `ID` stage, we see that the CPU now know which registers to fetch data from. However, the `writeRegister` and `writeData` signals only receive their values in the fifth cycle, or the `WB` stage.

Interestingly, `writeEnable` is activated earlier, However, this is a side effect of the fact the first instruction in the IM is `0x00000000`, which triggers an R-type control behavior. Nevertheless, this does not affect the program as only `0` is written into the `$zero` register. We also see the `writeEnable` signal deactivate in the 9th stage, which is the `WB` stage for the `sw` instruction.

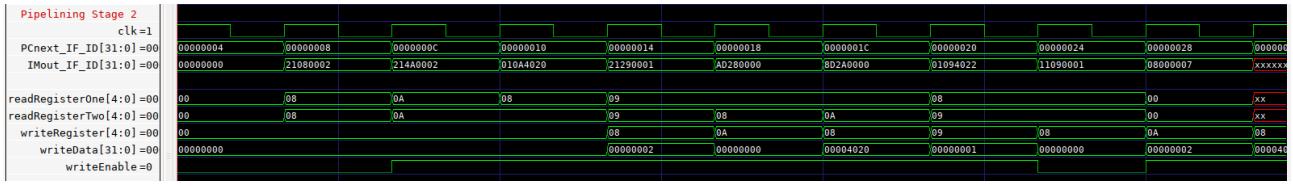

Figure 4: Wave output of stage 2

3.5 Pipelining Stage 3

The `PCnext-ID-EX` signal helps keep track of the instruction at which the behavior is occurring.

The first block shows that the ALU takes in the output of the register file directly. While the second block demonstrates that there is a mux selection between the register output and the sign-extended output for the ALU's second input.

There is also a mux-selection that occurs for the write register, which was moved to stage 3 for this implementation.

The muxes select appropriately according to the control signals from the pipeline register. For example, we see that `ALUin2` selects the register file output for the first two instructions (which are I-type), then selects the sign-extended output for the third instruction (R-type).

As a result, register `$t0` and `$t2` end up with value 2 in their respective `WB` stages. **However, the third instruction is supposed to update the value of register `$t0` to 4. In this implementation, because there is no data forwarding, nor is there a stall, register `$t0` is updated to value 0. This is the result of a data hazard, which is unsupported in this pipeline.**

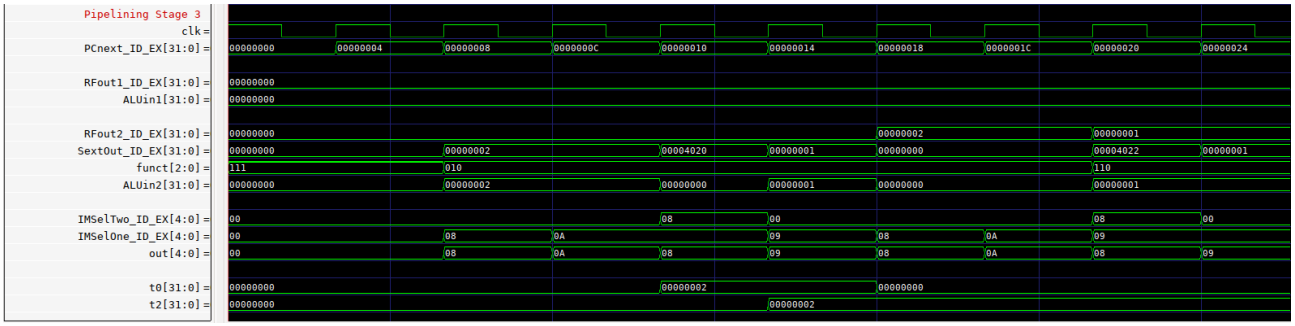

Figure 5: Wave output of stage 3

3.6 Pipelining Stage 4

In this stage, we are able to better see the reason why register `$t0` gets incorrectly updated. The `RFWriteReg-EX-MEM` signal carries register `$t0` at cycle 6. At this point, the `RFout2-EX-MEM` signal should be carrying the value 4; however, since register `$t2` write back hasn't been completed, this is not the case. The failure to update results in a cascading set of problems, where branching would be doomed because the value will never reach 1, which is the branch condition! The `BranchCtrl` signal shows that the instruction is a branch instruction, while the `Branch` signal shows whether this branch is obeyed or not. Since it is not obeyed, we see that the instructions linearly carry through to the next instruction.

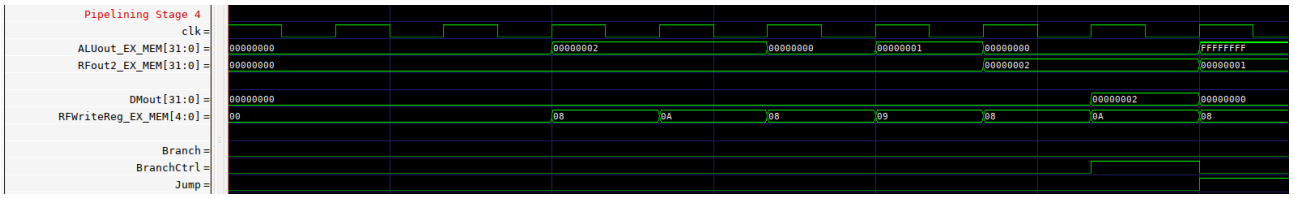

Figure 6: Wave output of stage 4

3.6.1 Introducing Jump

The provided diagram only allows support for branch instructions. Hence, the wiring was adjusted to allow for jumping. Originally, the `PCbranch-EX-MEM` signal and the `PCnext` signal is fed into a mux which decides the `PCin`.

To allow for jumps, the the `PCbranch-EX-MEM` signal and the `jumpAddress` signals are fed into a mux. The output of this mux is then fed into another mux along with `PCnext`, the output of which is fed into `PCin`. Thus, to support jump instructions, the implementation requires another mux.

The first mux selects for jump or branch addresses using the signal `Branch`. However, for the second mux, in this implementation, a new signal had to be created called PCSrc, which was an OR beteween `jump` and `branch`. This is because the `PCnext` signal must select for the signal that is *not* `PC + 4` if either jump or branch are enabled.

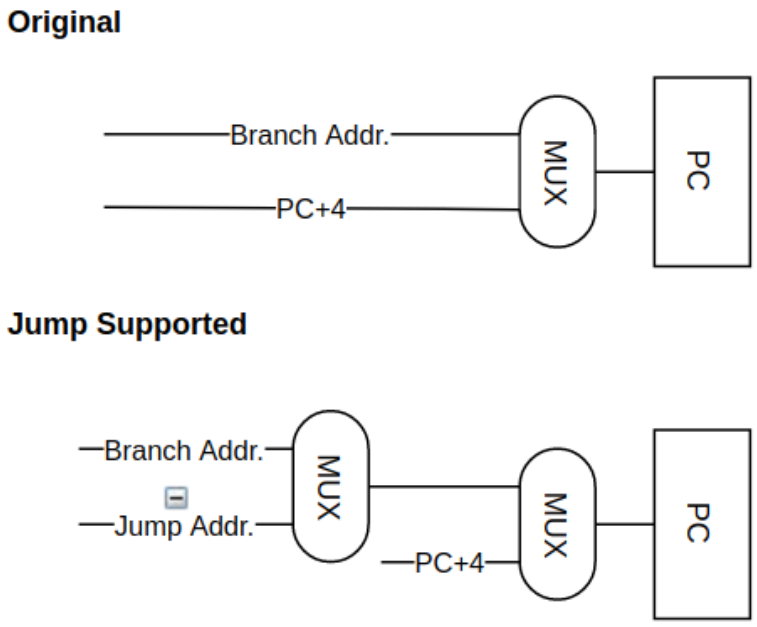

Figure 7: Diagram depicting jump instruction

Looking back at the signals, we see that the jump signal is enabled at the correct time. In our instructions, the jump instruction is responsible for carrying the loop back to a previous label.

The signals show that the jump instruction is selecting the correct address to jump to, but the program stops working shortly after. The `sub` instruction is correctly selected from the IM, confirming the jump instruction is actually carried out.

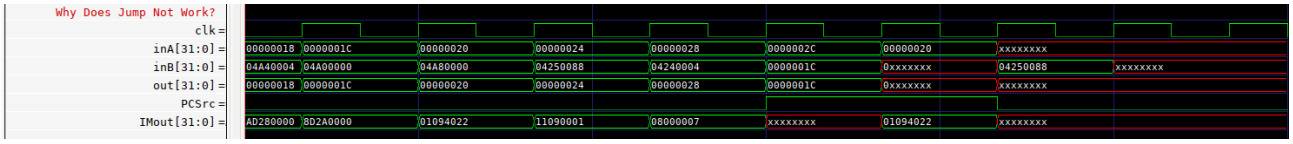

Figure 8: Diagram depicting why jump works

The data hazard which led to register `$t0` storing a value of 0, continues to cause a problem. The `sub` instruction now results in register `$t0` storing -1.

However, the root of the problem lies in the mux selecting a corrupted value for the next address, which in turn corrupts the output of the second mux as well, because it is set to the `beq` instruction! The screenshot below shows the inputs and output of the first mux selecting between the branch address and jump address. We see that in the 7th cycle, there is a corrupt output, which occurs because there is a gap in instruction selection.

The gap in instructions can be seen by the undefined `opcode` signal in between the defined `opcode` signals towards the end. **The gap occurs because the pipeline naturally takes in the instruction after the jump, while the jump is still being "computed". As a result of taking in a non-existent instruction, there are corruptions in signals. This could be best fixed by a stall after the jump to prevent taking in nonsense instructions.**

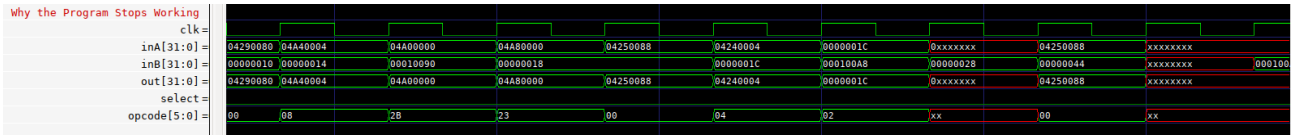

Figure 9: Diagram depicting why the program fails after jump

4 Conclusion

The two reasons why the program failed to run were the data hazard which ensured that register `$t0` would never have the correct output and the corrupt address selection because of naturally in-taking a non-existent instruction into the pipeline.